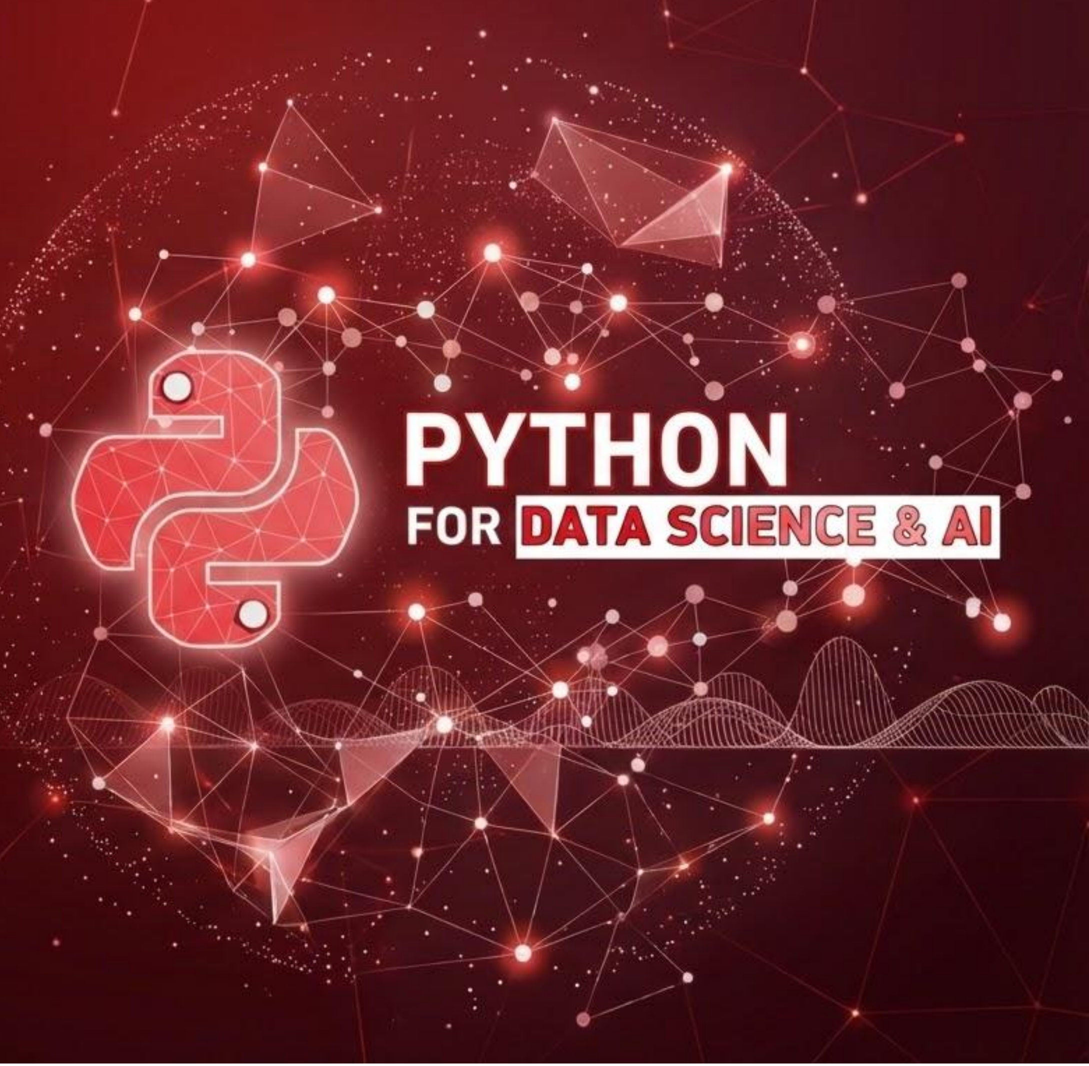
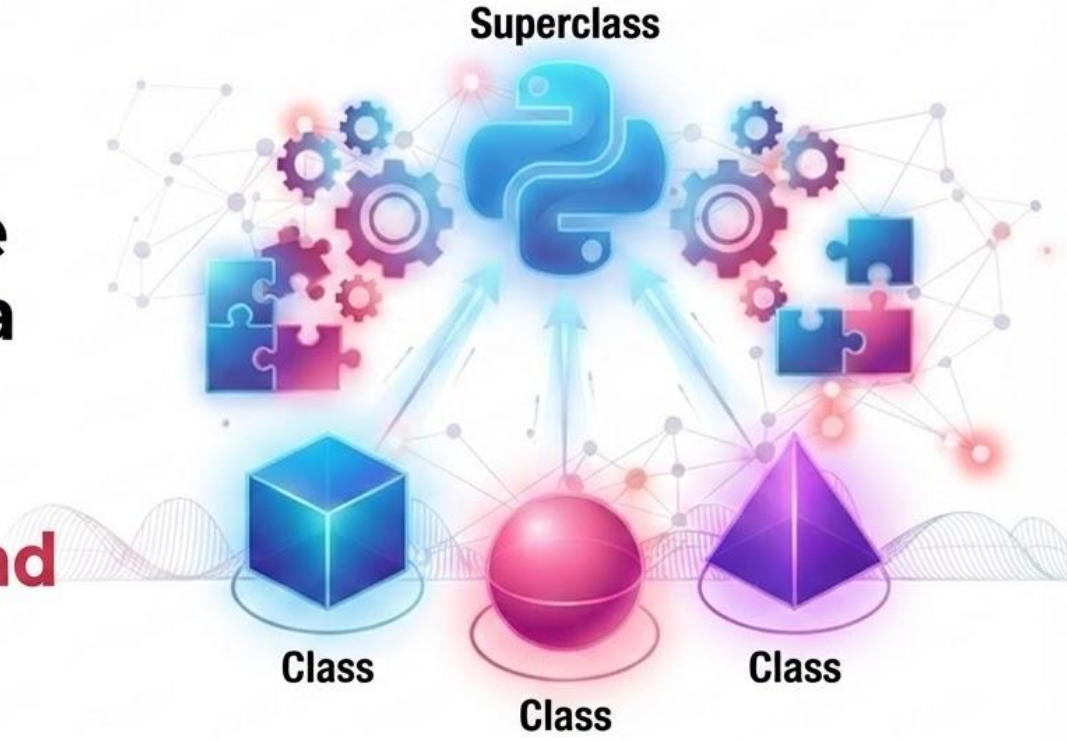
PYTHON FOR DATA SCIENCE & AI

Dasun Athukoralage
web: www.dasuna.me
email: dasun@nirvanaclouds.com



Polymorphism

It allows objects of different classes to be treated as objects of a common superclass, and it enables you to write more flexible and reusable code.



Polymorphism

This occurs when **a method has the same name in different classes**, but the **implementation of the method can be different for each class**.

This is achieved through **method overriding**, which allows a subclass to define a method with the **same name** as a method in its superclass.

Polymorphism

It means "**many forms**" – it allows the **same method call to behave differently** based on the object's actual class.

Draw Cube
Draw Sphere
Draw Pyramid
Draw Polygon

Polymorphism

Example:

```
class Polygon:
    # method to render a shape
    def render(self):
        print("Rendering Polygon...")

class Square(Polygon):
    # renders Square
    def render(self):
        print("Rendering Square...")

class Circle(Polygon):
    # renders circle
    def render(self):
          print("Rendering Circle...")
```

Polymorphism

```
# create an object of Square
s1 = Square()
s1.render()
# create an object of Circle
c1 = Circle()
c1.render()
```

```
#Output
Rendering Square...
Rendering Circle...
```

Encapsulation

———

Encapsulation means **hiding the internal details** of how an object works, and **only exposing what is necessary** to the outside.

You **bundle the data (attributes)** and the **methods (functions)** that work on that data **into a single unit – a class**.

Encapsulation

And you **protect the data** from **being directly accessed or changed.**

To prevent accidental change, an **object’s attributes can only be changed by an object’s method**.

Encapsulation

In Python, there are **no strict access modifiers** like in some other languages (**e.g., public, private, protected**). However, Python uses naming conventions to indicate the **intended access level**.

Encapsulation

---

1. **Public (no underscore):**

```
self.name = "John"
```

🔓 Can be accessed from anywhere — inside or outside the class.

Example:

```
obj.name  # Access is allowed
```

Encapsulation

---

2. Protected (single underscore _):

```
    self._name = "John"
```

🔓 Can be accessed **outside the class**, but by **convention** it’s intended **only for use inside the class or subclasses.**

Python does **not enforce** this — it’s a **“soft” protection.**

🧠 Think of it as: “You **can** access it, but **you shouldn’t unless you’re in a subclass.**”

Encapsulation

---

2. Protected (single underscore _):

Example:

```
class Person:

    def __init__(self):

        self._name = "Dasun"

class Employee(Person):

    def display(self):

        print(self._name)  # ✅ OK — accessed in subclass

e = Employee()

print(e._name)  # ❌ Technically allowed, but not recommended
```

Encapsulation

---

3. Private (double underscore __):

```
    self.__name = "John"
```

🔓 **Cannot be accessed directly** from outside the class.

🧠 Use **getter/setter methods** to access it.

Encapsulation

---

3. Private (double underscore __):

Example:

```
class Person:
    def __init__(self):
        self.__name = "John"
    def get_name(self):
        return self.__name
p = Person()
print(p.get_name())    # ✅ Good practice
print(p.__name)        # ❌ Error: AttributeError
```

Encapsulation

Getter & Setter methods:

To control access to private attributes, you can define getter and setter methods within the class. **Getter** methods **allow** you **to retrieve the value of a private attribute**, and **setter** methods **allow you to modify it**.

Encapsulation

```python
class BankAccount:
    def __init__(self):
        self.__balance = 0  # Private attribute

    def get_balance(self):
        return self.__balance

    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount

    def withdraw(self, amount):
        if amount > 0 and amount <= self.__balance:
            self.__balance -= amount
        else:
            print("Insufficient funds")
```

Encapsulation

Getter & Setter methods:

```
account = BankAccount()
account.deposit(1000)
print(account.get_balance())   # Accessing balance using getter
account.withdraw(500)
```

Benefits of Encapsulation

- **Data Hiding**: Encapsulation hides the internal details of a class, making it easier to manage and modify the class without affecting external code that uses the class.

- **Controlled Access**: It allows you to control access to attributes, ensuring that data is manipulated safely and consistently.

Benefits of Encapsulation

- **Modularity**: Encapsulation promotes modularity and code organization by encapsulating related data and behavior into a single unit (class).

- **Code Reusability**: Objects created from classes with encapsulation can be easily reused in different parts of your code or in other projects.

Example

Example 1:

Question: Please write an Object-Oriented Programming (OOP) code snippet in Python that models a bank account, **demonstrating encapsulation** by using **private attributes and methods.**

Example

Example 1:

```
class BankAccount:
    def __init__(self, account_number, initial_balance):
        self.__account_number = account_number
        self.__balance = initial_balance

    def __validate_amount(self, amount):
        return amount > 0

    def __update_balance(self, amount):
        if self.__validate_amount(amount):
            self.__balance += amount
```

Example

Example 1:

```
def __check_withdrawal(self, amount):
    return self.__validate_amount(amount) and amount <= self.__balance

def deposit(self, amount):
    self.__update_balance(amount)

def withdraw(self, amount):
    if self.__check_withdrawal(amount):
        self.__balance -= amount
```

Example

Example 1:

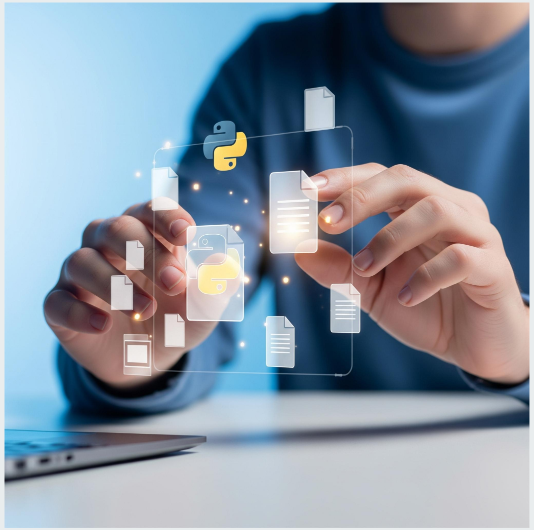
```
    def get_balance(self):
        return self.__balance

    def get_account_number(self):
        return self.__account_number
```

Example

Example 1:

```python
# Example usage:
account1 = BankAccount("123456", 1000.0)
print(f"Account Number: {account1.get_account_number()}")
print(f"Initial Balance: ${account1.get_balance()}")
account1.deposit(500.0)
#account1.__validate_amount(500.0) #This will generate an error
print(f"Balance after deposit: ${account1.get_balance()}")
account1.withdraw(200.0)
print(f"Balance after withdrawal: ${account1.get_balance()}")
```

File Handling



What is File Handling?

File handling in Python refers to the process of working with files, which can be used to **read data** from files, **write data** to files, and perform various operations on files.

What is File Handling?

Python provides **several built-in functions** and libraries for file handling, making it easy to work with files of different types, such as text files, binary files, and more.

File Operation

In Python, a file operation takes place in the following order:

1. **Open** a file

2. **Read** or **write** (perform operation)

3. **Close** the file

Opening files in Python

In Python, we use the **`open()`** method to open files.

To demonstrate how we open files in Python, let's suppose we have a file named **`my_file.txt`**.

Now, let's try to open data from this file using the **`open()`** function.

Opening files in Python

```
# open file in current directory

file_1 = open("my_file.txt")
```

Here, we have created a file object named file_1. This object can be used to work with files and directories.

Modes to Open a File

By default, the files are open in read mode (cannot be modified). The code in the previous slide is equivalent to

```
file1 = open("my_file.txt", "r")
```

Here, we have explicitly specified the mode by passing the **"r"** argument which means **file is opened for reading**.

Different Modes to Open a File

r	open a file for reading. (default)
w	Open a file for writing. Creates a new file if it does not exist or truncates the file if it exists.
x	Open a file for exclusive creation. If the file already exists, the operation fails.
a	Open a file for appending at the end of the file without truncating it. Creates a new file if it does not exist.
t	open in text mode. (default)

Different Modes to Open a File

b → Open in binary mode.

+ → Open a file for updating (reading and writing)

Reading Files in Python

After we open a file, we use the read() method to read its contents.

```
# open a file
file_1 = open("my_file.txt", "r")
# read the file
read_content = file_1.read()
print(read_content)
```

Closing Files in Python

When we are done with performing operations on the file, we need to properly close the file.

Closing a file will free up the resources that were tied with the file. It is done using the **`close()`** method in Python.

After we perform file operation, we should always close the file; it's a good programming practice.

The readline() method

---

readline() reads one line at a time from a file.

- Each call to readline() returns the next line in the file as a string.
- If the end of the file is reached, it returns an empty string ('').

The readline() method

```
file_1 = open("my_file.txt", "r")
line_1 = file_1.readline()
print(line_1)  #output: display the first line of the file.
line_2 = file_1.readline()
print(line_2)  #output: display the second line of the file.
file_1.close()
```

The readline() method

If you want to read all lines using readline() in a loop:

```
file = open("my_file.txt", "r")
while True:
    line = file.readline()
    if not line:
        break
    print(line.strip())  # strip() removes \n
file.close()
```

The readlines() method

---

readlines() reads **all the lines from a file** and returns them as a **list of strings**.

Each item in the list is **one line, including the newline character \n** at the end (unless it’s the last line and doesn’t have one).

The readlines() method

```
file = open("my_file.txt", "r")
lines = file.readlines()
print(lines)
file.close()
```

The readlines() method

Looping through readlines():

```
file = open("my_file.txt", "r")
lines = file.readlines()
for line in lines:
    print(line.strip())  # Removes \n

file.close()
```

Note: Don’t use readlines() on **very large files**, because it reads everything into memory at once. Use readline().

‘with’ statement for file handling in Python

```
with open("my_file.txt", "r") as file:
    content = file.read()
    print(content)
# No need for file.close() as it's automatically handled
```

Resource Management: Using **with** ensures that files are properly closed after operations are complete, preventing issues like file locks, resource leaks, or data corruption.

How to create files in Python

In Python, you use the open() function with one of the following options – "**x**" or "**w**" – to create a new file:

- **"x" – Create:** this command will **create a new file if and only if there is no file already in existence** with that name or **else it will return an error**.

```
file = open("new_file.txt", "x")
```

How to create files in Python

In Python, you use the open() function with one of the following options – "**x**" or "**w**" – to create a new file:

- **"w" – Write:** this command will create a new text file whether or not there is a file in the memory with the new specified name. **It does not return an error if it finds an existing file with the same name** – instead it will **overwrite the existing file**.

How to Write to a File in Python

There are two things we need to remember while writing to a file (In "**w**" mode).

- If we try to open a file that doesn't exist, a new file is created.
- If a file already exists, its content is erased, and new content is added to the file.

How to Write to a File in Python

In order to write into a file in Python, we need to open it in write mode by passing **"w"** inside open() as a second argument.

Suppose, we don't have a file named **my_file_2.txt**. Let's see what happens if we write contents to the **my_file_2.txt** file.

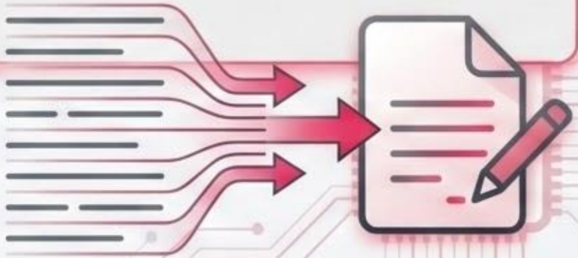
How to Write to a File in Python

```
with open("my_file_2.txt", "w") as file:
    file.write("This is awesome.\n")
    file.write("Programming is fun.\n")
```

The writelines() method

This function inserts multiple strings at the same time. A list of string elements is created, and each string is then added to the text file.

```
with open("test.txt", "w") as file:
    lines = ["This is something new.\n", "The writelines() method.\n"]
    file.writelines(lines)
```



Append data to an existing file

You can **append data** to an existing file using the **'a'** mode. Here’s how you can do it:

```
with open("test.txt", "a") as file:
    file.write("This is new data to append.\n")
```

Doing both read and write operations

If you want to open a file for both reading and writing, you can use the 'r+' or 'w+' mode. Here's how they work:

'r+' - Read and Write Mode:

- Opens the file for both reading and writing.
- The file must already exist; it raises an error if the file does not exist.
- Allows you to read and write data to the file.

Doing both read and write operations

If you want to open a file for both reading and writing, you can use the 'r+' or 'w+' mode. Here's how they work:

'r+' - Read and Write Mode:

- Does not truncate the file when opened.
- If you want to write to the end of the file, you'll need to seek to the end using the seek() method.
- Example: `file = open('example.txt', 'r+')`

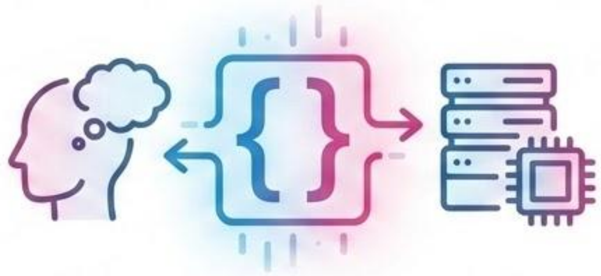
Doing both read and write operations

'w+' - Write and Read Mode:

- Opens the file for both reading and writing.
- Creates a new file if it doesn't exist.
- Truncates the file if it exists (removes its contents).
- Allows you to read and write data to the file.
- Example: `file = open('example.txt', 'w+')`

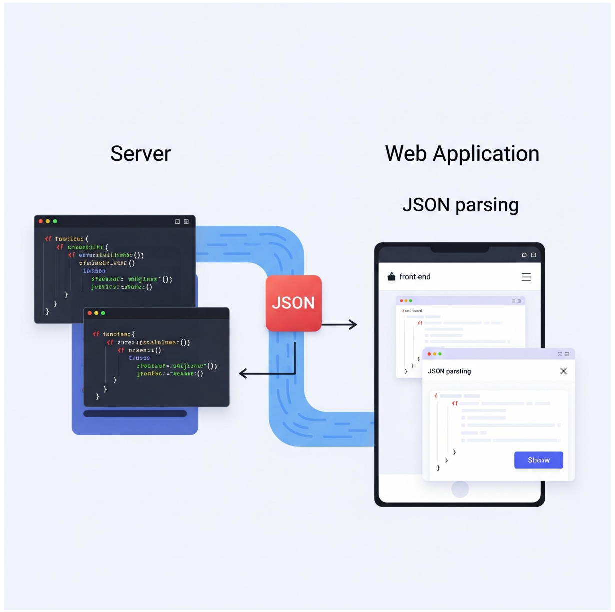
What is JSON?

JSON, which stands for **JavaScript Object Notation**, is a lightweight data interchange format that is easy for humans to read and write, and easy for machines to parse and generate.



What is JSON?

JSON is often used to **transmit data between a server and a web application**, or between different parts of an application.

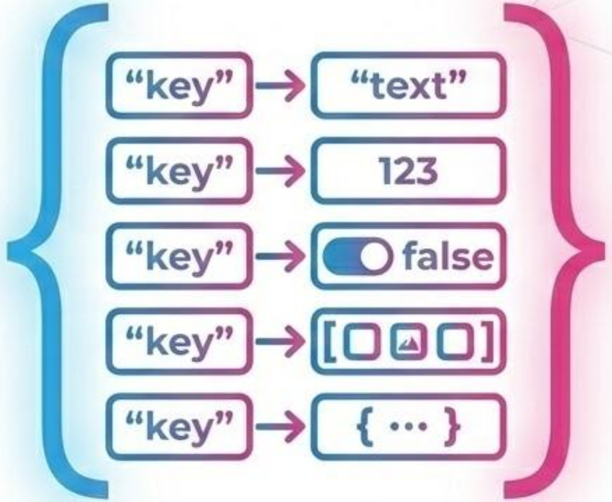


What is JSON?

It is a text-based format that is both **language-independent and platform-independent**, making it a popular choice for data exchange in a wide range of applications and programming languages.

What is JSON?

JSON data is represented as **a collection of key-value pairs**, where **keys are strings** enclosed in double quotation marks, and **values can be strings, numbers, booleans, arrays**, or other **JSON objects**.



What is JSON?

Example:

```
{
  "name": "John Doe",
  "age": 30,
  "isStudent": false,
  "hobbies": ["reading", "gardening", "swimming"]
```

```
{
  "name": "John Doe",
  "age": 30,
  "isStudent": false,
  "hobbies": ["reading", "gardening", "swimming"]
}
```

‘json’ module/library

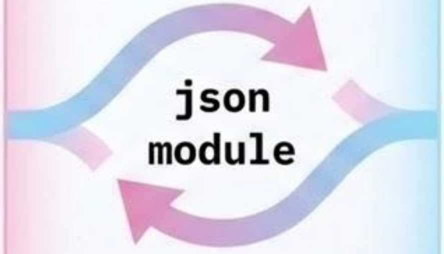
```
import json
```

The line "**import json**" in Python is used to import the **json module/library**, which provides functionality for working with **JSON** (**JavaScript Object Notation**) data.

‘json’ module/library

ENCODE (SERIALIZE)

Python data structures → JSON format

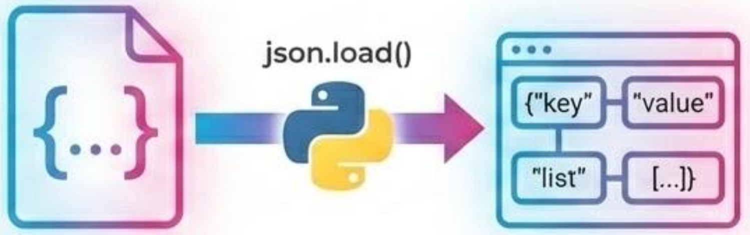


DECODE (DESERIALIZE)

JSON data → Python data structures

json.load() method

"json.load()" takes the JSON content from the file and turns it into something Python can understand and use – like a **dictionary** or a **list**.

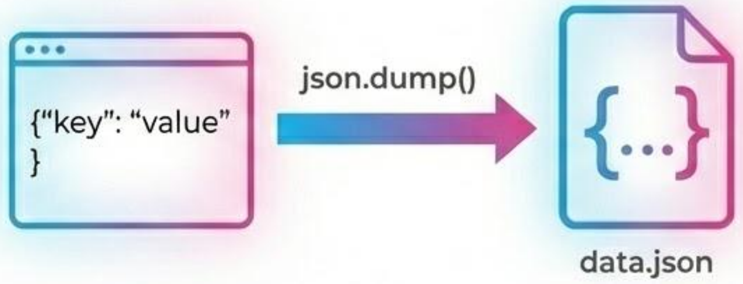


Reading JSON Data

```python
import json

# Specify the path to your JSON file
json_file_path = 'test.json'

# Open the JSON file for reading
with open(json_file_path, 'r') as json_file:
    # Load the JSON data into a Python dictionary
    data = json.load(json_file)

# Now 'data' contains the contents of the JSON file as a Python dictionary
print(data)
```

json.dump() method

json.dump() is used to save a Python object (like a dictionary) directly into a JSON file.
It **converts** the data to JSON format and **writes** it to the **file** – all in one step.

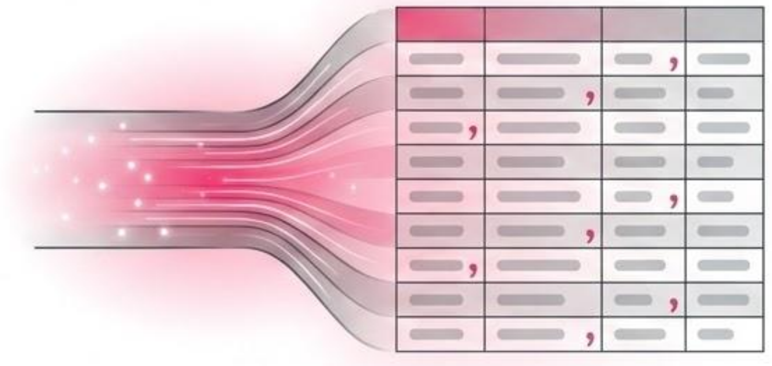


Writing JSON Data

```python
import json
# Create a Python dictionary
data = {
  "name": "John Nash",
  "age": 30,
  "city": "New York"
}
# Specify the path to the JSON file
json_file_path = 'my_json.json'
# Open the JSON file for writing
with open(json_file_path, 'w') as json_file:
  # Serialize the dictionary and write it directly to the file
  json.dump(data, json_file, indent=4)
print(f"JSON data has been written to {json_file_path}")
```

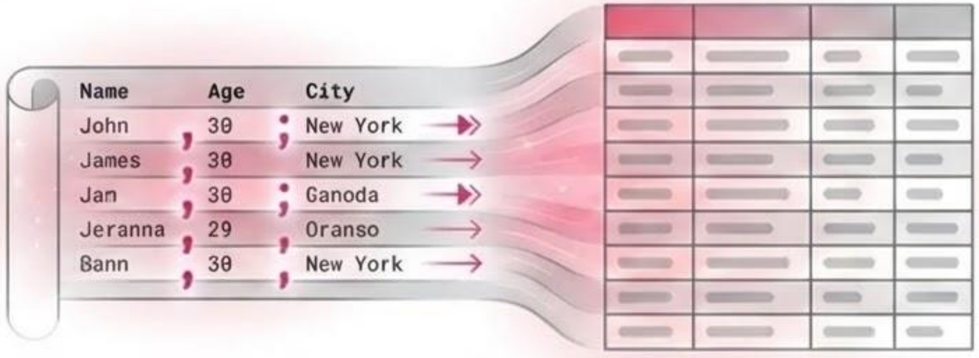
What is CSV?

CSV stands for **Comma-Separated Values**. It is a simple and widely used file **format** for storing **tabular data** (data organized in rows and columns).



What is CSV?

In a CSV file, each line typically represents a row of data, and the values within each row are separated by commas (or other delimiter characters, such as semicolons or tabs).



What is CSV?

Here's an example of what a CSV file might look like:

```
Name,Age,Occupation

John Doe,30,Engineer

Jane Smith,25,Teacher

Bob Johnson,40,Doctor

Alice Brown,35,Artist
```

‘csv’ module/library

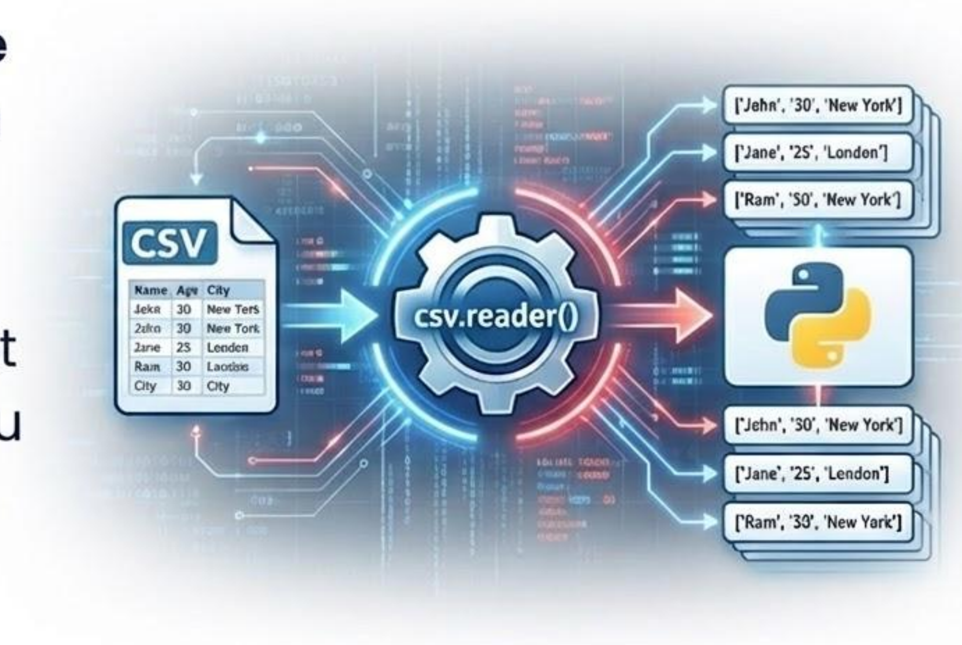
The csv module provides functionality for working with CSV (Comma-Separated Values) files, allowing you to read, write, and manipulate CSV data.

```python
import csv
```

csv.reader() method

- csv.reader() **reads a CSV file** and gives you each row as a **list of values**.

- csv.reader() is like a tool that **opens a CSV file** and lets you **read it line by line**, turning each row into a Python list.



Reading CSV Data

```
import csv

# Specify the path to the CSV file you want to read
csv_file_path = 'test.csv'

# Open the CSV file for reading
with open(csv_file_path, 'r', newline='') as csv_file:
  # Create a CSV reader object
  csv_reader = csv.reader(csv_file)

  # Iterate through each row in the CSV file
  for row in csv_reader:
      # 'row' is a list representing a row of data
      print(row)
```

Why newline='' ?

---

newline='' tells Python:

Don’t change newlines automatically – **let the csv module handle them properly**.

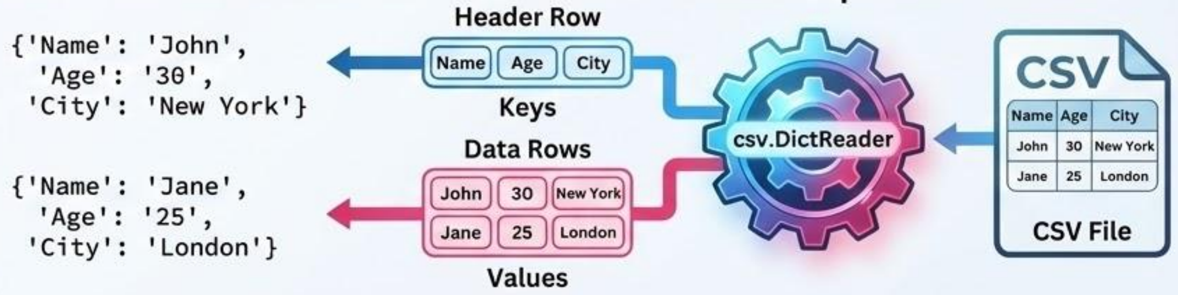
Reading CSV Data using DictReader

This is used to **read CSV files** into a **dictionary-like format.**

➢ Each row in the CSV file is read as a **dictionary** (**dict**).

➢ The **keys** of the dictionary are taken from the **header row** (the first row of the file).

➢ The **values** correspond to the data in each subsequent row.



```
{'Name': 'John',
  'Age': '30',
 'City': 'New York'}

{'Name': 'Jane',
  'Age': '25',
 'City': 'London'}
```

Reading CSV Data using DictReader

`csv.DictReader` makes it easier to work with CSV data because **you can access columns by their header names**, instead of using numeric indices.

Reading CSV Data using DictReader

```
import csv

# Specify the path to the CSV file you want to read
csv_file_path = 'test.csv'

# Open the CSV file
with open(csv_file_path, mode='r', newline='') as file:
  # Create a DictReader object
  reader = csv.DictReader(file)
   # Iterate through the rows
  for row in reader:
      print(row)  # Each row is a dictionary
```

Writing CSV Data

```
import csv

# Specify the path to the CSV file you want to create
csv_file_path = 'sample_output.csv'

# Data to be written to the CSV file (a list of dictionaries)
data = [
   {"Name": "John Doe", "Age": 30, "City": "New York"},
   {"Name": "Jane Smith", "Age": 28, "City": "Los Angeles"},
   {"Name": "Bob Johnson", "Age": 35, "City": "Chicago"}
]
# Define the CSV fieldnames (column names)
fieldnames = ["Name", "Age", "City"]
```

Writing CSV Data

```
# Open the CSV file for writing
with open(csv_file_path, 'w', newline='') as csv_file:
   # Create a CSV writer object
   csv_writer = csv.DictWriter(csv_file, fieldnames=fieldnames)

   # Write the header (column names) to the CSV file
   csv_writer.writeheader()
   # Write the data rows to the CSV file
   csv_writer.writerows(data)  # Write the data rows


print(f"CSV data has been written to {csv_file_path}")
```

Writing CSV Data

- We create a CSV writer object (**csv.DictWriter**) and provide it with the fieldnames.
- We use **csv_writer.writeheader()** to write the header (column names) to the CSV file.
- We use a for loop to iterate through the data list and write each row to the CSV file using **csv_writer.writerow(row)**

csv.DictWriter

Initialization: You create a csv.DictWriter object by providing the following parameters:

- A file-like object (e.g., an opened file) where the CSV data will be written.
- fieldnames: A list of column names that correspond to the keys in the dictionaries representing rows.
- Optional parameters such as delimiter, quotechar, and others that specify the CSV formatting options.

csv.DictWriter

Writing Data:

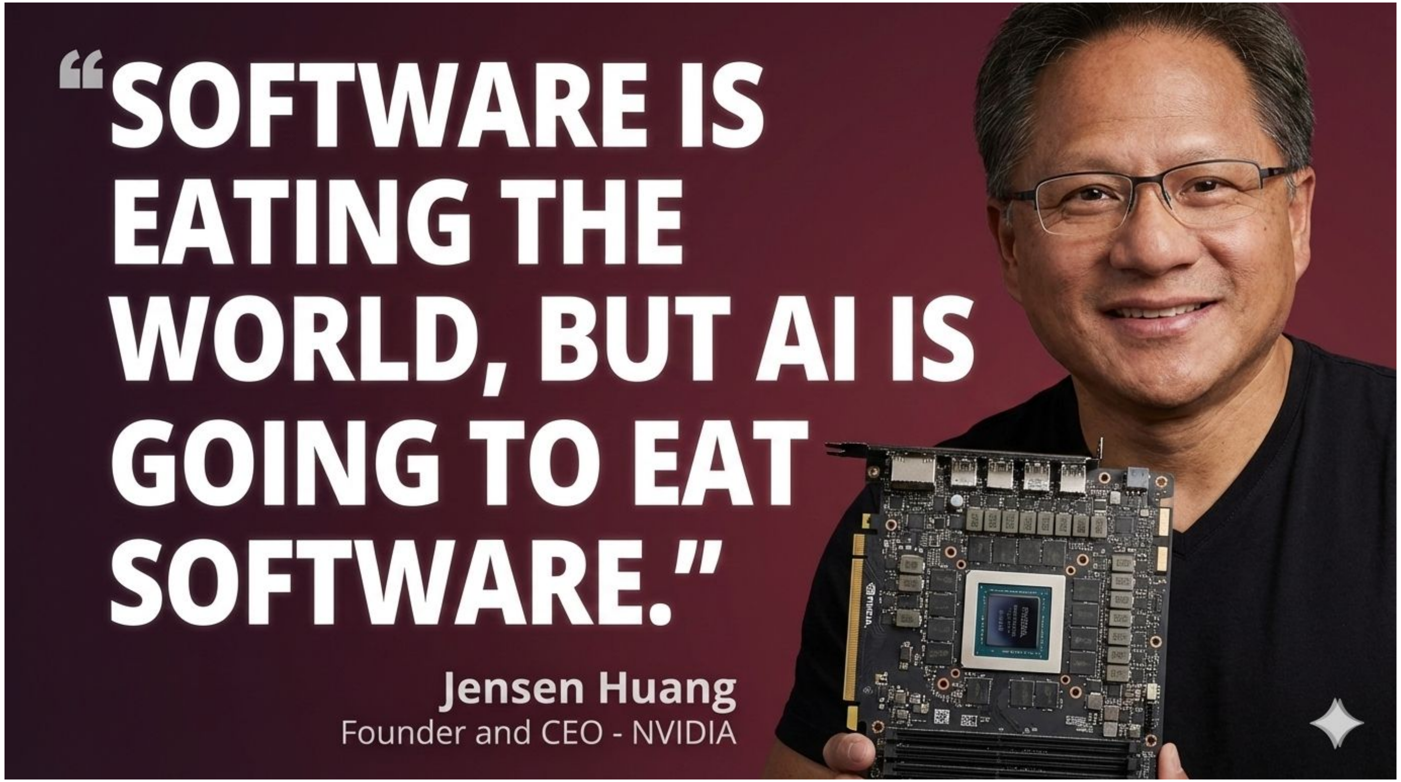
- You can use the **writeheader()** method to write the header row (column names) to the CSV file.
- You use the **writerow(dict)** method to write individual rows to the CSV file, where dict is a dictionary where the keys correspond to the column names.

csv.DictWriter

Automatic Matching: The csv.DictWriter class automatically matches the keys in the dictionary to the specified fieldnames when writing data to the file.

This means you don't have to worry about the order of columns; they will be written in the order you specified in fieldnames.

Thank You…!


"SOFTWARE IS EATING THE WORLD, BUT AI IS GOING TO EAT SOFTWARE."
Jensen Huang
Founder and CEO - NVIDIA

Thank You...!

Life is beautiful. Every day is a gift waiting to be unwrapped.